





**Commonly Used Lifecycle Methods**

The methods in this section cover the vast majority of use cases you’ll encounter creating React components. **For a visual reference, check out**[**this lifecycle diagram**](https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/)**.**

**render()**

render()

The render() method is the only required method in a class component.

When called, it should examine this.props and this.state and return one of the following types:

* **React elements.** Typically created via [JSX](https://legacy.reactjs.org/docs/introducing-jsx.html). For example, <div /> and <MyComponent /> are React elements that instruct React to render a DOM node, or another user-defined component, respectively.
* **Arrays and fragments.** Let you return multiple elements from render. See the documentation on [fragments](https://legacy.reactjs.org/docs/fragments.html) for more details.
* **Portals**. Let you render children into a different DOM subtree. See the documentation on [portals](https://legacy.reactjs.org/docs/portals.html) for more details.
* **String and numbers.** These are rendered as text nodes in the DOM.
* **Booleans or null or undefined**. Render nothing. (Mostly exists to support return test && <Child /> pattern, where test is boolean).

The render() function should be pure, meaning that it does not modify component state, it returns the same result each time it’s invoked, and it does not directly interact with the browser.

If you need to interact with the browser, perform your work in componentDidMount() or the other lifecycle methods instead. Keeping render() pure makes components easier to think about.

**Note**

render() will not be invoked if [shouldComponentUpdate()](https://legacy.reactjs.org/docs/react-component.html#shouldcomponentupdate) returns false.

**constructor()**

constructor(props)

**If you don’t initialize state and you don’t bind methods, you don’t need to implement a constructor for your React component.**

The constructor for a React component is called before it is mounted. When implementing the constructor for a React.Component subclass, you should call super(props) before any other statement. Otherwise, this.props will be undefined in the constructor, which can lead to bugs.

Typically, in React constructors are only used for two purposes:

* Initializing [local state](https://legacy.reactjs.org/docs/state-and-lifecycle.html) by assigning an object to this.state.
* Binding [event handler](https://legacy.reactjs.org/docs/handling-events.html) methods to an instance.

You **should not call setState()** in the constructor(). Instead, if your component needs to use local state, **assign the initial state to this.state** directly in the constructor:

constructor(props) {

super(props);

// Don't call this.setState() here!

this.state = { counter: 0 };

this.handleClick = this.handleClick.bind(this);

}

Constructor is the only place where you should assign this.state directly. In all other methods, you need to use this.setState() instead.

Avoid introducing any side-effects or subscriptions in the constructor. For those use cases, use componentDidMount() instead.

**Note**

**Avoid copying props into state! This is a common mistake:**

constructor(props) {

super(props);

// Don't do this!

this.state = { color: props.color };

}

The problem is that it’s both unnecessary (you can use this.props.color directly instead), and creates bugs (updates to the color prop won’t be reflected in the state).

**Only use this pattern if you intentionally want to ignore prop updates.** In that case, it makes sense to rename the prop to be called initialColor or defaultColor. You can then force a component to “reset” its internal state by [changing its key](https://legacy.reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html#recommendation-fully-uncontrolled-component-with-a-key) when necessary.

Read our [blog post on avoiding derived state](https://legacy.reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html) to learn about what to do if you think you need some state to depend on the props.

**componentDidMount()**

componentDidMount()

componentDidMount() is invoked immediately after a component is mounted (inserted into the tree). Initialization that requires DOM nodes should go here. If you need to load data from a remote endpoint, this is a good place to instantiate the network request.

This method is a good place to set up any subscriptions. If you do that, don’t forget to unsubscribe in componentWillUnmount().

You **may call setState() immediately** in componentDidMount(). It will trigger an extra rendering, but it will happen before the browser updates the screen. This guarantees that even though the render() will be called twice in this case, the user won’t see the intermediate state. Use this pattern with caution because it often causes performance issues. In most cases, you should be able to assign the initial state in the constructor() instead. It can, however, be necessary for cases like modals and tooltips when you need to measure a DOM node before rendering something that depends on its size or position.

**componentDidUpdate()**

componentDidUpdate(prevProps, prevState, snapshot)

componentDidUpdate() is invoked immediately after updating occurs. This method is not called for the initial render.

Use this as an opportunity to operate on the DOM when the component has been updated. This is also a good place to do network requests as long as you compare the current props to previous props (e.g. a network request may not be necessary if the props have not changed).

componentDidUpdate(prevProps) {

// Typical usage (don't forget to compare props):

if (this.props.userID !== prevProps.userID) {

this.fetchData(this.props.userID);

}

}

You **may call setState() immediately** in componentDidUpdate() but note that **it must be wrapped in a condition** like in the example above, or you’ll cause an infinite loop. It would also cause an extra re-rendering which, while not visible to the user, can affect the component performance. If you’re trying to “mirror” some state to a prop coming from above, consider using the prop directly instead. Read more about [why copying props into state causes bugs](https://legacy.reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html).

If your component implements the getSnapshotBeforeUpdate() lifecycle (which is rare), the value it returns will be passed as a third “snapshot” parameter to componentDidUpdate(). Otherwise this parameter will be undefined.

**Note**

componentDidUpdate() will not be invoked if [shouldComponentUpdate()](https://legacy.reactjs.org/docs/react-component.html#shouldcomponentupdate) returns false.

**componentWillUnmount()**

componentWillUnmount()

componentWillUnmount() is invoked immediately before a component is unmounted and destroyed. Perform any necessary cleanup in this method, such as invalidating timers, canceling network requests, or cleaning up any subscriptions that were created in componentDidMount().

You **should not call setState()** in componentWillUnmount() because the component will never be re-rendered. Once a component instance is unmounted, it will never be mounted again.

**Rarely Used Lifecycle Methods**

The methods in this section correspond to uncommon use cases. They’re handy once in a while, but most of your components probably don’t need any of them. **You can see most of the methods below on**[**this lifecycle diagram**](https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/)**if you click the “Show less common lifecycles” checkbox at the top of it.**

**shouldComponentUpdate()**

shouldComponentUpdate(nextProps, nextState)

Use shouldComponentUpdate() to let React know if a component’s output is not affected by the current change in state or props. The default behavior is to re-render on every state change, and in the vast majority of cases you should rely on the default behavior.

shouldComponentUpdate() is invoked before rendering when new props or state are being received. Defaults to true. This method is not called for the initial render or when forceUpdate() is used.

This method only exists as a [**performance optimization**](https://legacy.reactjs.org/docs/optimizing-performance.html)**.** Do not rely on it to “prevent” a rendering, as this can lead to bugs. **Consider using the built-in**[**PureComponent**](https://legacy.reactjs.org/docs/react-api.html#reactpurecomponent) instead of writing shouldComponentUpdate() by hand. PureComponent performs a shallow comparison of props and state, and reduces the chance that you’ll skip a necessary update.

If you are confident you want to write it by hand, you may compare this.props with nextProps and this.state with nextState and return false to tell React the update can be skipped. Note that returning false does not prevent child components from re-rendering when *their* state changes.

We do not recommend doing deep equality checks or using JSON.stringify() in shouldComponentUpdate(). It is very inefficient and will harm performance.

Currently, if shouldComponentUpdate() returns false, then [UNSAFE\_componentWillUpdate()](https://legacy.reactjs.org/docs/react-component.html#unsafe_componentwillupdate), [render()](https://legacy.reactjs.org/docs/react-component.html#render), and [componentDidUpdate()](https://legacy.reactjs.org/docs/react-component.html#componentdidupdate) will not be invoked. In the future React may treat shouldComponentUpdate() as a hint rather than a strict directive, and returning false may still result in a re-rendering of the component.

**static getDerivedStateFromProps()**

static getDerivedStateFromProps(props, state)

getDerivedStateFromProps is invoked right before calling the render method, both on the initial mount and on subsequent updates. It should return an object to update the state, or null to update nothing.

This method exists for [rare use cases](https://legacy.reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html#when-to-use-derived-state) where the state depends on changes in props over time. For example, it might be handy for implementing a <Transition> component that compares its previous and next children to decide which of them to animate in and out.

Deriving state leads to verbose code and makes your components difficult to think about. [Make sure you’re familiar with simpler alternatives:](https://legacy.reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html)

* If you need to **perform a side effect** (for example, data fetching or an animation) in response to a change in props, use [componentDidUpdate](https://legacy.reactjs.org/docs/react-component.html#componentdidupdate) lifecycle instead.
* If you want to **re-compute some data only when a prop changes**, [use a memoization helper instead](https://legacy.reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html#what-about-memoization).
* If you want to **“reset” some state when a prop changes**, consider either making a component [fully controlled](https://legacy.reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html#recommendation-fully-controlled-component) or [fully uncontrolled with a key](https://legacy.reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html#recommendation-fully-uncontrolled-component-with-a-key) instead.

This method doesn’t have access to the component instance. If you’d like, you can reuse some code between getDerivedStateFromProps() and the other class methods by extracting pure functions of the component props and state outside the class definition.

Note that this method is fired on *every* render, regardless of the cause. This is in contrast to UNSAFE\_componentWillReceiveProps, which only fires when the parent causes a re-render and not as a result of a local setState.

**getSnapshotBeforeUpdate()**

getSnapshotBeforeUpdate(prevProps, prevState)

getSnapshotBeforeUpdate() is invoked right before the most recently rendered output is committed to e.g. the DOM. It enables your component to capture some information from the DOM (e.g. scroll position) before it is potentially changed. Any value returned by this lifecycle method will be passed as a parameter to componentDidUpdate().

This use case is not common, but it may occur in UIs like a chat thread that need to handle scroll position in a special way.

A snapshot value (or null) should be returned.

For example:

class ScrollingList extends React.Component {

constructor(props) {

super(props);

this.listRef = React.createRef();

}

getSnapshotBeforeUpdate(prevProps, prevState) {

// Are we adding new items to the list?

// Capture the scroll position so we can adjust scroll later.

if (prevProps.list.length < this.props.list.length) {

const list = this.listRef.current;

return list.scrollHeight - list.scrollTop;

}

return null;

}

componentDidUpdate(prevProps, prevState, snapshot) {

// If we have a snapshot value, we've just added new items.

// Adjust scroll so these new items don't push the old ones out of view.

// (snapshot here is the value returned from getSnapshotBeforeUpdate)

if (snapshot !== null) {

const list = this.listRef.current;

list.scrollTop = list.scrollHeight - snapshot;

}

}

render() {

return (

<div ref={this.listRef}>{/\* ...contents... \*/}</div>

);

}

}

In the above examples, it is important to read the scrollHeight property in getSnapshotBeforeUpdate because there may be delays between “render” phase lifecycles (like render) and “commit” phase lifecycles (like getSnapshotBeforeUpdate and componentDidUpdate).

**Error boundaries**

[Error boundaries](https://legacy.reactjs.org/docs/error-boundaries.html) are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed. Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

A class component becomes an error boundary if it defines either (or both) of the lifecycle methods static getDerivedStateFromError() or componentDidCatch(). Updating state from these lifecycles lets you capture an unhandled JavaScript error in the below tree and display a fallback UI.

Only use error boundaries for recovering from unexpected exceptions; **don’t try to use them for control flow.**

For more details, see [*Error Handling in React 16*](https://legacy.reactjs.org/blog/2017/07/26/error-handling-in-react-16.html).

**Note**

Error boundaries only catch errors in the components **below** them in the tree. An error boundary can’t catch an error within itself.

**static getDerivedStateFromError()**

static getDerivedStateFromError(error)

This lifecycle is invoked after an error has been thrown by a descendant component. It receives the error that was thrown as a parameter and should return a value to update state.

class ErrorBoundary extends React.Component {

constructor(props) {

super(props);

this.state = { hasError: false };

}

static getDerivedStateFromError(error) { // Update state so the next render will show the fallback UI. return { hasError: true }; }

render() {

if (this.state.hasError) { // You can render any custom fallback UI return <h1>Something went wrong.</h1>; }

return this.props.children;

}

}

**Note**

getDerivedStateFromError() is called during the “render” phase, so side-effects are not permitted. For those use cases, use componentDidCatch() instead.

**componentDidCatch()**

componentDidCatch(error, info)

This lifecycle is invoked after an error has been thrown by a descendant component. It receives two parameters:

1. error - The error that was thrown.
2. info - An object with a componentStack key containing [information about which component threw the error](https://legacy.reactjs.org/docs/error-boundaries.html#component-stack-traces).

componentDidCatch() is called during the “commit” phase, so side-effects are permitted. It should be used for things like logging errors:

class ErrorBoundary extends React.Component {

constructor(props) {

super(props);

this.state = { hasError: false };

}

static getDerivedStateFromError(error) {

// Update state so the next render will show the fallback UI.

return { hasError: true };

}

componentDidCatch(error, info) { // Example "componentStack": // in ComponentThatThrows (created by App) // in ErrorBoundary (created by App) // in div (created by App) // in App logComponentStackToMyService(info.componentStack); }

render() {

if (this.state.hasError) {

// You can render any custom fallback UI

return <h1>Something went wrong.</h1>;

}

return this.props.children;

}

}

Production and development builds of React slightly differ in the way componentDidCatch() handles errors.

On development, the errors will bubble up to window, this means that any window.onerror or window.addEventListener('error', callback) will intercept the errors that have been caught by componentDidCatch().

On production, instead, the errors will not bubble up, which means any ancestor error handler will only receive errors not explicitly caught by componentDidCatch().

**Note**

In the event of an error, you can render a fallback UI with componentDidCatch() by calling setState, but this will be deprecated in a future release. Use static getDerivedStateFromError() to handle fallback rendering instead.